COMP 302 Winter 2025 Problem Set 3

Define

```
type 'a susp = Susp of (unit -> 'a)
let force (Susp f) = f ()

type 'a stream = { hd: 'a; tl: 'a stream susp }
```

Problem 1: Power Series

Create a lazy stream representing the power series $\sum_{i=0}^{\infty} \frac{1}{2^i}$.

```
let rec power_series index x =
    (* Implement *)
```

Problem 1 Solution

```
let rec power_series index x = {
  hd = x;
  tl = Susp (fun () ->
    let new_term = 1. /. (2. ** float_of_int index) in
    power_series (index + 1) (x + new_term))
}
```

Problem 2: Infinite List

Convert a regular list to an infinite list by repeating its elements indefinitely

```

cycle : 'a list -> 'a stream

let rec cycle list =
    (* Implement *)
```

Problem 2 Solution

```

let rec cycle list =
  let rec inner cycle_list =
    match cycle_list with
    | [] -> inner list
    | h::t -> { hd = h; tl = Susp (fun () -> inner t) }
  in
  inner list
```

Problem 3: Triple Fibonacci Sequence Generator

Create a function that generates a "Triple Fibonacci" sequence, where each term is the sum of the last three terms, starting with initial values.

```
let rec triple_fib a b c =
    (* Implement *)
```

Problem 3 Solution

```
let rec triple_fib a b c = {
  head = a;
  tail = Susp (fun () -> {
    head = b;
    tail = Susp (fun () -> {
      head = c;
      tail = Susp (fun () -> triple_fib b c (a + b + c))
    })
  })
}
```

Problem 4: Sorted Merge

Given two streams each containing integers, merge them into a single sorted stream without duplicates.

```
let rec merge_sorted s1 s2 =
  (* Implement * )
```

Problem 4 Solution

```
let rec merge_sorted s1 s2 =
  match (s1.hd, s2.hd) with
  | (x, y) when x = y ->
      { hd = x; tl = Susp (fun () -> merge_sorted (force s1.tl) (force s2.tl)) }
  | (x, y) when x < y ->
```

```
      { hd = x; tl = Susp (fun () -> merge_sorted (force s1.tl) s2) }
  | (x, y) ->
      { hd = y; tl = Susp (fun () -> merge_sorted s1 (force s2.tl)) }
```

Problem 5: Change Making

Given the following implementation of the change-making problem using continuations in OCaml

```
(* Semi-CPS change-making with an accumulator *)
let rec change coins amt acc fail = match coins, amt with
  | _, 0 -> acc
  | [], _ -> fail ()
  | c::cs, amt when amt >= c ->
      change (c::cs) (amt-c) (c::acc) (fun () -> change cs amt acc fail)
  | c::cs, amt ->
      change cs amt acc fail
```

Transform this backtracking solution into a lazy stream generator that produces all possible ways to make change.

```
type 'a susp = Susp of (unit -> 'a)
type 'a lazy_list = { hd : 'a; tl : 'a fin_list susp }
and 'a fin_list = Empty | NonEmpty of 'a lazy_list

let rec go_gen_change coins amt acc next =
    (*Implement*)

let gen_change coins amt : int list fin_list =
    (*Implement* )
```

Problem 5 Solution

```
let rec go_gen_change coins amt acc next =
    match coins, amt with
    | _, 0 -> NonEmpty { hd = acc; tl = Susp next }
    | [], _ -> next ()
    | c::cs, amt ->
        go_gen_change (c::cs) (amt-c) (c::acc)
            (fun () -> go_gen_change cs amt acc next)


let gen_change coins amt : int list fin_list =
    go_gen_change coins amt [] (fun () -> Empty)
```